A Dynamic Filtering Algorithms to Search Approximate String

N.Rajasekhar
Associate professor
Dayananda Sagar College of Engineering, Bangalore
Mail id rajasekhar531@gmail.com

Abstract: Here we consider the problem of finding the relative strings those are comparative to given query string. For that various similarity\distance measures can be used like total divergence to average[15],Bhattacharya coefficient[16],cosine similarity[17]jaccard coefficient[18], and Levenshtein distance[19] This problem is of great interests to a variety of applications that need a high real-time performance, such as data cleaning, query relaxation, and spellchecking. We identify existing indexes, search algorithms, filtering strategies, selectivity-estimation techniques and other work, and comment on their respective merits and limitations.. in this paper we propose several algorithms that can greatly improve the performance of existing algorithms. Second, Filtering is a standard technique for fast approximate string matching in practice. In filtering, a quick first step is used to rule out almost all positions of a text as possible starting positions for a pattern. Typically this step consists of finding the exact matches of small parts of the pattern and for that we propose consolidation of existing filtering techniques with these algorithms, and show that they should be used assembled into one smartly, since the way to do the consolidation can greatly affect the performance. We have conducted experiments on several real data sets to evaluate the proposed techniques

I. Introduction

Text data is ubiquitous. Management of string data in databases and information systems has taken on particular importance recently. In this paper, we study the following problem: given a collection of strings, how to efficiently find those in the collection that are similar to a query string? Such a query is called an “approximate string search.” This problem is of great interests to a variety of applications, as illustrated by the following examples.

Spell Checking: Given an input document, a spellchecker needs to find possibly mistyped words by searching in its dictionary those words similar to the words. Thus, for each word that is not in the dictionary, we need to find potentially matched candidates to recommend.

Data Cleaning: Information from different data sources often have various inconsistencies. The same real-world entity could be represented in slightly different formats. There could also be errors in the original data introduced in the data-collection process. For these reasons, data cleaning needs to find from a collection of entities those similar to a given entity. A typical query is “find addresses similar to PO Box 23, Main St.”, and an entity of “P.O. Box 23, Main St” should be found and returned.

These applications require a high real-time performance for each query to be answered, especially for those applications adopting a Web-based service model. For instance, consider a spellchecker such as those used by Gmail, Hotmail, or Yahoo Mail. It needs to be invoked many times every second since there can be millions of users of the service. Each spellchecking request needs to be processed as fast as possible. Although from an individual user’s perspective, there is not much difference between a 20ms processing time and a 2ms processing time, from the server’s perspective, the former means 50 queries per second (QPS), while the latter means 500 queries per second. Clearly the latter processing time gives the server more power to serve more user requests per second. Thus it is very important to develop algorithms for answering such queries as efficiently as possible.

Many algorithms have been proposed, such as [1], [2], [3], [4], [5], [6], [7]. These techniques assume a given similarity function to quantify the closeness between two strings. Different string-similarity functions have been studied, such as edit distance [8], cosine similarity [2], and Jaccard coefficient [9]. Many of these algorithms use the concept of *gram*, which is a substring of a string to be used as a signature of the string. These algorithms rely on inverted lists of grams to find candidate strings, and utilize the fact that similar strings should share enough common grams. Many algorithms [9], [2] mainly focused on “join queries,” i.e., finding similar pairs from two collections of strings. Approximate string search could be treated as a special case of join queries. It is well understood that the behavior of an algorithm for answering selection queries could be very different from that for answering join queries. We believe approximate string search is important enough to deserve a separate investigation.

Our contributions: In this paper we make two main contributions. First, we propose three efficient algorithms for an-swering approximate string search queries, called ScanCount, Merge Skip, and Divide Skip. Normally, the main operation in answering such queries is to merge the inverted lists of the grams produced from the query string. The ScanCount algorithm adopts a simple idea of scanning the inverted lists and counting candidate strings. Despite the fact that it is very naive, when combined with various filtering techniques, this algorithm can still achieve a high performance. The MergeSkip algorithm exploits the value differences among the inverted lists and the threshold on the number of common grams of similar strings to skip many irrelevant candidates on the lists. The DivideSkip algorithm combines the MergeSkip algorithm and the idea in the MergeOpt algorithm proposed in [9] that divides the lists into two groups. One group is for those long lists, and the other group is for the remaining lists. We run the MergeSkip algorithm to merge the short lists with a different threshold, and use the long lists to verify the candidates. Our experiments on three real data sets showed that the proposed algorithms could significantly improve the performance of existing algorithms.

Our second contribution is a study on how to integrate various filtering techniques with the proposed merging algo-rithms. Various filters have been proposed to eliminate strings that cannot be similar enough to a given string. Surprisingly, our experiments and analysis show that a naive solution of adopting all available filtering techniques might not achieve the best performance to merge inverted lists. Intuitively, filters can segment inverted lists to relatively shorter lists, while merging algorithms need to merge these lists. In addition, the more filters we apply, the more groups of inverted lists we need to merge, and more overhead we need to spend for processing these groups before merging their lists. Thus filters and merging algorithms need to be integrated judiciously by considering this tradeoff. Based on this analysis, we classify filters into two categories: *single-signature filters* and *multi-signature filters*. We propose a strategy to selectively choose proper filters to build an index structure and integrate them with merging algorithms. Experiments show that our strategy reduces the running time by as much as one to two orders of magnitude over approaches without filtering techniques or strategies that naively use all the filtering techniques.

In this paper we consider several string similarity functions, including edit distance, Jaccard, Cosine, and Dice [2]. We qualify the effectiveness and generalization capability of these techniques by showing our new merging and filtering strategies are efficient for those similarity functions.

Paper Outline: Section II gives the preliminaries. Section III presents our new algorithms. Section IV discusses how to judiciously integrate filtering techniques with merging algo-rithms. Section V shows how the results on edit distance can be extended to other similarity functions. Section VI discusses related work, and Section VII concludes this paper.

II. Preliminaries

Let Σ be an alphabet. For a string s of the characters in Σ, we use "$|s|$" to denote the length of s, "$s[i]$" to denote the i-th character of s (starting from 1), and "$s[i, j]$" to denote the substring from its i-th character to its j-th character.

Q-Grams: We introduce two characters α and β not in Σ. Given a string s and a positive integer q, we extend s to a new string s by prefixing $q-1$ copies of α and suffixing $q-1$ copies of β. A *positional q-gram* of s is a pair (i, g), where g is the q-gram of s starting at the i-th character of s, i.e., $g = s[i, i+q-1]$. The set of *positional q-grams* of s, denoted by $G(s, q)$ (or simply $G(s)$ when the q value is clear in the context) is obtained by sliding a window of length q over the characters of string s. There are $|s| + q - 1$ positional q-grams
in $G(s, q)$. For instance, suppose $\alpha = \#$, $\beta = \$$, $q = 3$, and $s =$ "smith", then $G(s, q) = \{$(1, ##s), (2, #sm), (3, smi),
(4, mit), (5, ith), (6, th$), (7, h$$)$\}$. Our discussion in this paper is also valid when strings are not extended using the special characters.

Approximate String Search: Given a collection of strings S, a query string Q, and a threshold δ, we want to find all $s \in S$ such that the similarity between s and Q is no less than δ. Various similarity functions can be used, such as edit distance, Jaccard similarity, cosine similarity, and dice similarity. In this paper, we first focus on edit distance, then generalize our techniques to other similarity functions. The *edit distance* (a.k.a. Levenshtein distance) between two strings s_1 and s_2 is the minimum number of edit operations of single characters that are needed to transform s_1 to s_2. Edit operations include insertion, deletion, and substitution.
We denote the edit distance between two strings s_1 and s_2 as $ed(s_1, s_2)$. For example, ed("Steven Spielburg", "Steve Spielberg") = 2. When using this function, our problem becomes finding all $s \in S$ such that $ed(s, Q) \leq k$ for a given threshold k.

III. Merging Algorithms

Several existing algorithms assume an index of inverted lists for the grams of the strings in the collection S to answer approximate string queries on S. In the index, for each gram g of the strings in S, we have a list l_g of the ids of the strings that include this gram, possibly with the corresponding positional information of the gram in the strings. It is observed in [9] that the search problem based on several string-similarity functions can be solved by solving the following generalized problem:

> *T-occurrence Problem*: Let Q be a query, and $G(Q, q)$ be its corresponding set of q-grams for a constant q. Find the set of string ids that appear at least T times on the inverted lists of the grams in $G(Q, q)$, where T is a constant.

For instance, it is known that if the edit distance be-tween two strings s_1 and s_2 is no greater than k, then they should share at least the following number of q-grams: $T = max\{|s_1|, |s_2|\} + q - 1 - k \cdot q$. If this threshold is zero or negative, then we need to scan the entire data set in order to compute the answers. Various filters can help us reduce the number of strings that need to be scanned (Section IV).

The result of generalized problem is a set of candidate strings. We then need to eliminate the false positives in the candidates by applying the similarity function on the candi-dates and the query string. Existing algorithms for solving this problem focus on reducing the running time to merge the record-id (RID) lists of the grams of the query string. An established optimization is to sort the record ids on each inverted list in an ascending order. We briefly describe two existing algorithms as follows [9].

Heap algorithm: When merging the lists, we maintain the frontiers of the lists as a heap. At each step, we pop the top from the heap, and increment the count of the record id corresponding to the popped frontier record. We remove this record id from this list, and reinsert the next record id on the list (if any) to the heap. We report a record id whenever its count is at least the threshold T. Let $N = |G(Q, q)|$ denote the number of lists corresponding to the grams from the query string, and M denote the total size of these N lists. This algorithm requires $O(M \log N)$ time and $O(N)$ storage space (not including the size of the inverted lists) for storing the heap of the frontiers of the lists.

MergeOpt algorithm: It treats the $T - 1$ longest inverted lists of $G(Q, q)$ separately. For the remaining $N - (T - 1)$ relatively short inverted lists, we use the Heap algorithm to merge them with a lower threshold, i.e., 1. For each candidate string, we do a binary search on each of the $T - 1$ long lists to verify if the string appears on at least T times among all the lists. This algorithm is based on the observation that a record in the answer must appear on at least one of the short lists. Experiments have shown that the algorithm is significantly more efficient than the Heap algorithm.

We now present three new merging algorithms.

A. *Algorithm: ScanCount*

This algorithm improves the Heap algorithm by eliminating the heap data structure and the corresponding operations on the heap. Instead, we just maintain an array of counts for all the string ids in S. We scan the N inverted lists one by one. For each string id on each list, we increment the count corresponding to the string by 1. We report the string ids that appear at least T times on the lists. The algorithm is formally descried in Figure 1.

```
Input: set of RID lists and a threshold $T$;
Output: record ids that appear at least $T$ times on the lists.
1. Initialize the array $C$ of $|S|$ counters to 0's;
2. Initialize a result set $R$ to be empty;
3. FOR (each record id $r$ on each given list) {
4.     Increment the value of $C[r]$ by 1;
5.     IF ($C[r]$ == $T$)
6.         Add $r$ to $R$;
7. }
8. RETURN $R$;
```

Fig. 1. ScanCount Algorithm.

The time complexity of this algorithm is $O(M)$ (compared to $O(M \log N)$ for the Heap algorithm). The space complexity is $O(|S|)$, where $|S|$ is the size of the string collection, since we need to keep a count for each string id. This higher space complexity (compared to $O(N)$ for the Heap algorithm) is not a major concern, since this extra space tends to much smaller than that of the inverted lists. This algorithm shows that the T-occurrence problem is indeed different from the problem of merging multiple sorted lists into one long sorted lists, since we care more about finding those ids with enough occurrences, rather than generating a sorted list.

One computational overhead in the algorithm is the step to initialize the counter array to 0's for each query (line 1). This step can be eliminated by storing an additional query id for each counter in the array. When the system first gets started, we initialize the counters to 0's, and their associated query ids to be 0. When a new query arrives, we assign a unique id to the query (incrementally from 0). Whenever we access the counter for a string id from an inverted list, we first check if the query id associated with the counter is the same as the current query id. If so, we take the same actions as before. Otherwise, we assign the new query id to this string id, and set its counter to 1. In this way, we do not need to initialize the array counters for each query. The drawback of this new approach is that in each iteration, we need to do an additional comparison of the two query ids. Which approach is more efficient depends on the total number of strings in the collection, the expected number of strings whose counters need to be updated, and whether we want to support multiple queries concurrently.

Despite its simplicity, this algorithm could still achieve a good performance when combined with various filtering techniques, if they can shorten the inverted lists to be merged, as shown in our experimental results.

B. *Algorithm: MergeSkip*

This algorithm is formally described in Figure 2. Its main idea is to skip on the lists those record ids that cannot be in the answer to the query, by utilizing the threshold T. Similar to Heap algorithm; we also maintain a heap for the frontiers of these lists. A key difference is that, during each iteration, we pop those records from the heap that have the same value as the top record t on the heap. Let the number of popped records be n. If there are at least T such records, we add t to the result set (line 8 in the algorithm), and add their next records on the lists to the heap. Otherwise, we are sure record t cannot be in the answer. In addition to popping these n records, we pop $T - 1 - n$ additional records from the heap (line 12). Therefore, in this case, we have popped $T - 1$ records from the heap. Let t be the current top record on the heap. For each of the $T - 1$ popped lists, we locate its smallest record r such that $r \geq t$ (line 15). This locating step can be done efficiently using a binary search. We then push r to the heap (line 16). Notice that it is possible to reinsert the same record on the popped lists back to the heap if it is equal to the new top record t. Also for those lists that do not have such a record $r \geq t$, we do not insert any record from these lists to the heap.

As an example, consider the four RID lists shown in Figure 3 and a threshold $T = 3$. At the beginning, we push their frontier ids 1, 10, 50, and 100 to the heap. The current top of the heap is id 1. There is only one record on the heap with the value, and we pop this record from the heap (i.e., $n = 1$). Then we pop $T - 1 - n = 3 - 1 - 1 = 1$ smallest record from the heap, which is the record id 10 (line 12 in the algorithm). Now the top record on the heap is $t = 50$, as shown on the right-hand side in the figure. For each of the two popped lists, we locate the next record (using a binary search) that is no greater than 50. In this way, we can skip many records that cannot be in the answer. The next records on these two lists both have the same value 50. In the next iteration, we have three records with the current top-record value 50, and we add this record to the result set.

C. *Algorithm: DivideSkip*

Its main idea is to combine MergeSkip and MergeOpt, both of which try to skip irrelevant records on the lists,

```
Input: a set of RID lists and a threshold $T$;
Output: record ids that appear at least $T$ times on the lists.
1.  Insert the frontier records of the lists to a heap $H$;
2.  Initialize a result set $R$ to be empty;
3.  WHILE ($H$ is not empty) {
4.      Let $t$ be the top record on the heap;
5.      Pop from $H$ those records equal to $t$;
6.      Let $n$ be the number of popped records;
7.      IF ($n \geq T$ ) {
8.          Add $t$ to $R$;
9.          Push next record (if any) on each popped list to $H$;
10.     }
11.     ELSE {
12.         Pop $T - 1 - n$ smallest records from $H$;
13.         Let $t$ be the current top record on $H$;
14.         FOR (each of the $T - 1$ popped lists) {
15.             Locate its smallest record $r \geq t$ (if any);
16.             Push this record to $H$;
17.         }
18.     }
19. }
20. RETURN $R$;
```

Fig. 2. MergeSkip Algorithm.

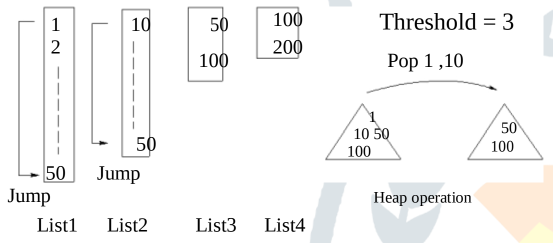


Fig. 3. Running MergeSkip algorithm.

but using different intuitions. MergeSkip exploits the *value differences* among the records on the lists, while MergeOpt exploits the *size differences* among the lists. Our new algorithm DivideSkip uses both differences to further improve the search performance.

Figure 4 formally describes the algorithm. Given a set of RID lists, we first sort these lists based on their lengths. We divide the lists into two groups. We group the L longest lists to a set L_{long} , and the remaining short lists as another set L_{short}. (The choice of the parameter L will be discussed shortly.) We use the MergeSkip algorithm on L_{short} to find records r that appear at least $T - L$ times on the short lists. For each such record r and each list l_{long} in L_{long} , we check if r appears on l_{long} . This step can be done efficiently using $O(logp)$ time (where p is the length of l_{long}) if the list is implemented as an ordered list, or $O(1)$ time if the list is implemented as an unordered hash set. If the total number of occurrences of r among all these lists is at least T , then we add it to the result set R.

There are two main differences between MergeOpt and DivideSkip. (1) The number of long lists in DivideSkip is a tunable parameter L, which can greatly affect the performance of the algorithm. In MergeOpt, L is fixed to a constant $T - 1$. (2) Unlike MergeOpt, which uses a heap-based algorithm

```
Input: set of RID lists and a threshold $T$;
Output: record ids that appear at least $T$ times on the lists.
1.  Initialize a result set $R$ to be empty;
2.  Let $L_{long}$ be the set of $L$ longest lists among the lists;
3.  Let $L_{short}$ be the remaining short lists;
4.  Use MergeSkip on $L_{short}$ to find ids that appear at
      least $T - L$ times;
5.  FOR (each record $r$ found) {
6.      FOR (each list in $L_{long}$)
7.          Check if $r$ appears on this list;
8.      IF ($r$ appears $\geq T$ times among all lists)
9.          Add $r$ to $R$;
10. }
11. RETURN $R$;
```

Fig. 4. DivideSkip Algorithm.

to process L_{short}, the DivideSkip algorithm uses the more efficient MergeSkip algorithm to process the short lists.

1) Choosing Parameter L in DivideSkip: The parameter L affects the overall performance of the algorithm in two ways. If we increase L, fewer lists are treated as short lists, which need to be merged with a lower threshold $T - L$. The time of accessing the short lists will decrease. On the other hand, for each candidate after accessing the short lists, we need to do more lookups on the long lists. A main issue is how to choose a good L value for this algorithm. The best L value is difficult to decide since it depends on the query and its inverted lists. We propose a formula to calculate a good value for the parameter L for a given query, which has been empirically shown to be a close-to-optimal value.

Proposition 1: Given a set of inverted lists and a threshold T , a good L value in DivideSkip can be estimated as:

$$L_{good} = \frac{T}{\mu logM + 1}, \quad (1)$$

where M is the length of the longest inverted list of the grams of the query, and μ is a coefficient dependent on the data set, but independent from the query.

The following is the intuition behind this formula. Let L denote the number of lists in L_{long} , and N denote the total number of records in L_{short}. The total time to access the short lists can be estimated as:

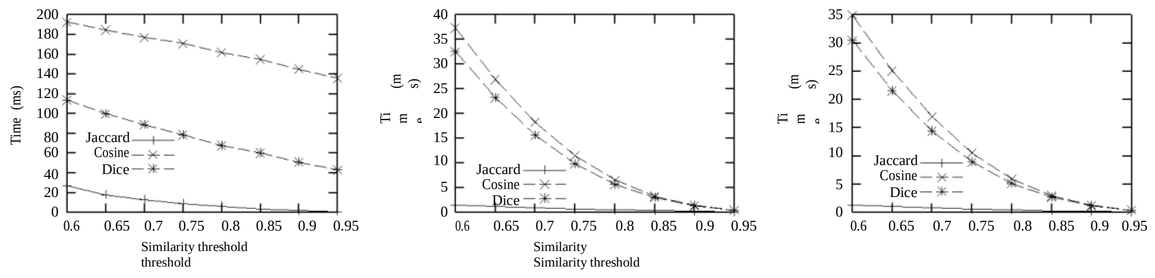
$$C_1 = \varphi \cdot N, \quad (2)$$

where φ is a constant. Let x denote the number of records whose number of occurrences in L_{short} is at least $\geq T - L$. We can estimate x as $\frac{\eta \cdot N}{T - L}$, where η is a parameter dependent on the data set S. For each candidate record from the short lists, its lookup time in the long lists can be estimated as $L \cdot logM$. Hence, the total lookup time on the long lists is:

$$C_2 = \frac{\eta \cdot N}{T - L} \cdot L \cdot \log M. \quad (3)$$

The total running time is $C_1 + C_2$. There is a tradeoff between C_1 and C_2. Assuming that the best performance is achieved when $C_1 = C_2$, we can get Equation 1 by replacing $\frac{\eta}{\varphi}$ by μ.

The parameter μ in the formula can be computed offline as follows. We generate a workload of queries. For each query q_i , we try different L values and identify its optimal value for this query.



(a) No filter.
filter.
filters.

(b) Length
(c) Length and prefix

Fig. 14. Running time of DivideSkip using different similarity functions (DBLP data set).

search queries are important enough to deserve a separate investigation, which is the focus of this paper.

Recently, Kim et al. [14] proposed a technique called "n-Gram/2L" to improve space and time efficiency for inverted index structures. Li et al. [5] proposed a new technique called VGRAM to judiciously choose high-quality grams of variable lengths from a collection of strings. Our research in this paper is orthogonal to these studies and complementary to their work on grams. Our merging algorithms are independent on the indexing strategy, and can be easily used by those variant techniques based on grams.

VII. CONCLUSION

In this paper we studied how to efficiently find in a collection of strings those similar to a given string. We made two contributions. First, we developed new algorithms that can greatly improve the performance of existing algorithms. Second, we studied how to integrate existing filtering techniques with these algorithms, and showed that they should be used together judiciously, since the way to do the integration can greatly affect the performance. We reported the results of our extensive experiments on several real data sets to evaluate the proposed techniques.

REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik, "Efficient Exact Set-Similarity Joins," in *VLDB*, 2006, pp. 918–929.
[2] R. Bayardo, Y. Ma, and R. Srikant, "Scaling up all-pairs similarity search," in *WWW Conference*, 2007.
[3] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning," in *SIGMOD*, 2003, pp. 313–324.
[4] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001, pp. 491–500.
[5] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *Very Large Data Bases*, 2007.
[6] E. Sutinen and J. Tarhio, "On Using q-Grams Locations in Approximate String Matching," in *ESA*, 1995, pp. 327–340.
[7] E. Ukkonen, "Approximae String Matching with q-Grams and Maximal Matching," *Theor. Comut. Sci.*, vol. 1, pp. 191–211, 1992.
[8] V. Levenshtein, "Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones," *Profl. Inf. Transmission*, vol. 1, pp. 8–17, 1965.
[9] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicate," in *ACM SIGMOD*, 2004.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006, pp. 5–16.
[11] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.
[12] K. Ramasamy, J. M. Patel, R. Kaushik, and J. F. Naughton, "Set containment joins: The good, the bad and the ugly," in *VLDB*, 2000.
[13] N. Koudas, S. Sarawagi, and D. Srivastava, "Record linkage: similarity measures and algorithms," in *SIGMOD Tutorial*, 2005, pp. 802–803.
[14] Learning Subject Areas by Using Unsupervised Observation of Most Informative Terms in Text Databases,Tangudu Naresh, G.Ramesh Naidu, S.Vishnu Murty,ijera volume2,issue1,jan-feb2012 pp.1044-49
[15] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee, "n-Gram/2L: A space and time efficient two-level n-gram inverted index structure." in *VLDB*, 2005, pp. 325–336.
[16] http://en.wikipedia.org/wiki/Bhattacharyya_distance
[17] Similarity Measures for Text Document Clustering, Anna Huang Department of Computer Science,The University of Waikato, Hamilton, New Zealand, *NZCSRSC 2008*, April 2008
[18] Tan, Pang-Ning; Steinbach, Michael; Kumar, Vipin (2005), *Introduction to Data Mining*, ISBN 0-321-32136-7
[19] http://en.wikipedia.org/wiki/Levenshtein_distance